Refactoring for Immutability

Fredrik Berg Kjolstad Danny Dig Gabriel Acevedo Marc Snir
University of Illinois
Urbana-Champaign, USA
{kjolsta1,dig,acevedo7,snir}@illinois.edu

Abstract

It is common for Object-Oriented programs to contain mutable and immutable classes. Immutable classes simplify sequential programing because the programmer does not have to reason about side-effects. Immutable classes also simplify parallel programming because they are embarrassingly thread-safe. Sometimes programmers write immutable classes from scratch, other times they refactor mutable into immutable classes. To refactor a mutable class, programmers must find (i) all methods that mutate its transitive state and (ii) all objects that can enter or escape the state of the class. The programmer must also update client code to use the class in an immutable fashion. The analysis is non-trivial and the rewriting is tedious. Fortunately, this can be automated.

We present IMMUTATOR, a technique and tool that enables the programmer to safely refactor a mutable class into an immutable class. IMMUTATOR also repairs client code to use the refactored class in an immutable fashion. Experience with refactoring several open-source programs shows that IMMUTATOR is useful: (i) it reduces the burden of making classes immutable, (ii) is fast enough to be used interactively, and (iii) is much safer than manual refactorings performed by open-source developers.

1. Introduction

An immutable object is one whose state cannot be mutated after the object has been initialized and returned to a client. By object state we mean the *transitively reachable state*: the state of the object and all state reachable from that object by following references. Immutability has long been touted as one of the features that makes functional programming an excellent fit for both sequential and parallel programming [27].

Immutability can make sequential programs *simpler* and more *efficient*. An immutable object (also known as a *value object* [21]) is simpler to reason about [11] because there are no side-effects, and is simpler to debug. Immutable objects facilitate persistent storage [4], they are good hash-table keys [15], they can be compared very efficiently by identity comparison [4], they can reduce memory footprint (through interning/memoization [16, 19] or flyweight [12]). Immutable objects also enable several compiler optimizations (e.g., reducing the number of dynamic reads [20]). In fact, some argue that we should always use immutable classes unless we explicitly need mutability [6].

Technical Report For early dissemination of ideas.

Immutability can also simplify parallel programming [13, 18]. Since threads can not change the state of an immutable object, it can be shared among threads without any synchronization. An immutable object is *embarrassingly thread-safe* and as with any embarrassingly parallel problems, we should take advantage and not be embarrassed.

Immutability also simplifies distributed programming [4]. With the current middleware technologies like Java RMI, EJB, or Corba, a client can send messages to a distributed object via local proxies, which must implement the updating protocol. If an object is immutable, there is no need to keep a proxy.

Having all objects immutable is not advisable either. When immutable objects contain lots of data, memory consumption can become an issue. Any attempt to update the object requires copying the object state, which in turn creates lots of memory churn.

It is therefore common for OO programs to contain both mutable and immutable objects, thus combining the strengths of both approaches. Objects that are updated frequently are mutable objects, thus they do not incur the cost of copying their entire state upon a mutation. Objects that are seldomly updated can be made immutable and copied upon rare mutations. The programmer, who is expert on the problem domain, knows which objects should be made immutable and which ones should remain mutable.

Mainstream OO languages like Java, C++, and C# favor mutability by default. Although they have support for shallow, non-transitive immutability through keywords such as `final`, `readonly`, and `const`, they do not have support for deep, transitive immutability. Of the three languages, C++ has a stronger notion of transitive immutability. Notice that in C++, `const` can be applied to a complete object as long as the object is not allocated on the heap. However, this only applies to the local state of the object (i.e., the stack-allocated state), and not to other objects reachable through pointers. That is, a pointer in a `const` object can still be used to mutate its referent.

To get the strongest immutability guarantees, the immutability must be `built-in` the class. If a class is *immutable*, none of its instances can be (transitively) mutated. Java includes many immutable classes, for example `String` and most classes in the `Number` class hierarchy (e.g., `Integer`). Sometimes programmers write an immutable class from scratch, other times they refactor a mutable class into an immutable class.

To refactor a mutable Java class (from here on referred as the *target class*) into an immutable class, the programmer needs to perform several tasks. First, she must declare all fields as `final`, meaning that they can not be assigned outside of constructors and field initializers. However, declaring a field as `final` is not enough. In Java, this only makes the *reference* immutable, not the object that is pointed to by the field (i.e., it is a *shallow*, not a *deep* immutability [5]). In other words, it suffices to declare scalar, primitive types as final, but for object and array types this is not enough because the *transitive state* of those fields can still be mutated. The program-

mer must search through the methods of the target class and find all the places where the transitive state is mutated. This task is further complicated by aliases, mutations nested deep inside call chains, and polymorphic methods. Moreover, the programmer needs to ensure that objects do not escape out of the target class (e.g., through return statements), where they can be mutated by clients of the target class. Finding the escaping objects is not trivial either. For example, an object can be added to a container class (e.g., a `List`), and then the method can return the container along with the escaped object.

Furthermore, once the programmer has found all mutations, she must decide how to handle them. She can delete the mutating method, re-implement it to throw an exception, or re-implement the method as a factory method that returns a new object. She must also find and handle objects entering or escaping the class, for example by cloning those objects.

These code rewritings required changing several lines of code per target class in the open-source projects that we studied. The analysis for finding the target class mutations and class escapes is non-trivial and error-prone, and the code rewriting can be tedious.

To alleviate the programmer's burden when refactoring mutable into immutable classes, we designed and implemented IMMUTATOR, a technique and tool for making Java classes immutable. IMMUTATOR rewrites the target class by replacing mutating methods with factory methods that return a new object whose state is the old state plus the mutation. IMMUTATOR rewrites the client code to use the target class in an immutable fashion. For example, IMMUTATOR rewrites `target.mutateMethod()` to `target=target.mutateMethod()`.

We implemented IMMUTATOR on top of Eclipse's refactoring engine. It therefore offers all the convenience of a modern refactoring tool such as previewing changes, preserving format and comments, and undo. To use it the programmer selects a target class and chooses "Make Immutable" from the refactoring menu. IMMUTATOR then analyzes whether the refactoring is safe. If the refactoring is safe, then it rewrites the code. However, if one of the preconditions are not met, it warns the programmer and provides information about the problem.

At the heart of IMMUTATOR are three analyses that determine the safety of the refactoring. The first analysis is an inter-procedural analysis that determines which methods mutate the transitively-reachable state of the target class. The second analysis is a class escape analysis that detects whether objects that are a part of the target class state *may* escape. The third analysis uses a context-sensitive, demand-driven pointer analysis [25] to detect aliases of variables on which mutating methods are invoked.

There is a large body of work on detecting side-effect free methods [3, 22–24] and on escape analysis [7, 29]. The previous work has a *general scope*. Previous mutation analyses were designed to detect any side-effects, including mutations to method arguments that are not part of the transitive state of the target class. Similarly, previous escape analyses were designed to detect any objects that escape a method, including local variables that are not part of the transitive state of the target class. In contrast, our analyses have a more *focused scope*: we are only interested in detecting mutator methods that update the transitive state of the target class. An immutable class can still have methods with side-effects on other parts of the heap, but not on the transitive state of the target class. Similarly, we are only interested to detect escaping objects that are part of the transitive state of the target class. Although our analyses are using constructs similar with those of existing analyses, their search scope is different enough to prevent us from plugging existing analyses.

This paper makes the following contributions:

1. **Problem Description.** While there are many systems for specifying and checking immutability, this is the first paper describing the problems and challenges that arise when refactoring a mutable into an immutable class.
2. **Analyses.** We present three interprocedural analyses to check refactoring preconditions. One analysis determines the mutating methods, another determines class escape, and another determines aliases. Although we are building our analyses on top of established program analyses, we are presenting new usages of older analyses, and our search scope differs. Also, we are presenting on-demand analyses that are efficient enough to be used in an interactive refactoring tool.
3. **Transformations.** We present the transformations needed for an automated tool to convert a Java class to an immutable class.
4. **Tool.** We have implemented the analyses as well as the code transformations in an automated refactoring tool, IMMUTATOR, integrated with the Eclipse IDE.
5. **Evaluation.** We designed a controlled experiment where the participants manually refactored 9 JHotDraw classes, then we refactored the same classes with IMMUTATOR. We also conducted case-studies of how open-source developers refactored immutable classes. When comparing the manually and IMMUTATOR-refactored code, we found out that IMMUTATOR outperforms developers: refactoring with IMMUTATOR is *safer* and *faster*.

IMMUTATOR as well as the experimental evaluation data can be downloaded from: `http://refactoring.info/tools/Immutator`

2. Motivating Example

As our running example, we use class `Circle`, shown on the left-hand side of Fig. 1. `Circle` has a center, stored in the `c` field, and a radius, stored in its `r` field. There are several methods to modify or retrieve the state. At the bottom of the figure we show a client that creates a circle and prints its state. The programmer decides to refactor this class into an immutable class, since it makes sense to treat mathematical objects as value objects.

Refactoring even a simple class like `Circle` into an immutable class (see right-hand side of Fig. 1) is non-trivial. First, the programmer must find all the mutating methods. Method `setRadius` (line 19) is a direct mutator because it directly changes a field and it is fairly easy to spot due to the field assignment (and so is `moveTo(Point)`. Method `moveTo(int, int)` (line 27) is a mutator too. However, the code in line 29 does not change the field reference `c`, instead it changes the object that `c` points to. This method thus changes the transitive state of `Circle`. Method `moveBy` (line 33) is a mutator too, but it does not mutate the state directly. Instead, it mutates state indirectly by invoking the direct mutator `moveTo(Point)`. In general, finding all mutators (indirect and transitive) is complicated by arbitrary long method call chains, aliases, and polymorphic methods.

Furthermore, the programmer must locate all the places where an object enters or escapes the target class. Consider the client code that creates a `center` point (line 46) and passes it to the `circle`. Even if the target class was immutable, since the client class holds a reference to the `center` object, the client can still mutate it from outside of the target class. Since we cannot foresee or access all uses of the target class, we must conservatively assume that the target class can be mutated through entering and escaping objects. Coming back to the target class, the programmer must find all the places where objects enter in the target class (line 23–24), or escape (line 40).

Finally, by making the target class immutable, the programmer is changing the semantics of the class itself. Therefore, she has to change the client code to compensate for this: she must refactor the client code to use the target class in an immutable fashion (see lines 48–49).

```
1  public class Circle {
2    private Point c = new Point(0, 0);
3    private int   r = 1;
4
5
6
7
8
9
10
11
12
13
14
15   public int getRadius() {
16     return r;
17   }
18
19   public void setRadius(int r) {
20     this.r = r;
21   }
22
23   public void moveTo(Point p) {
24     this.c = p;
25   }
26
27   public void moveTo(int x, int y) {
28
29     c.setLocation(x, y);
30
31   }
32
33   public void moveBy(int dx, int dy) {
34     Point newCenter = new Point(c.x + dx, c.y + dy);
35     moveTo(newCenter);
36
37   }
38
39   public Point getLocation() {
40     return c;
41   }
42 }
43
44 class Client {
45   public static void main(String[] args) {
46     Point center = new Point(1, 1);
47     Circle circle = new Circle();
48     circle.setRadius(7);
49     circle.moveTo(center);
50     System.out.print("center=" + circle.getLocation() +
51                      ", radius=" + circle.getRadius());
52   }
53 }
```

```
1  public final class Circle {
2    private final Point c;
3    private final int   r;
4
5    public Circle() {
6      this.c = new Point(0, 0);
7      this.r = 1;
8    }
9
10   private Circle(Point c, int r) {
11     this.c = c;
12     this.r = r;
13   }
14
15   public int getRadius() {
16     return r;
17   }
18
19   public Circle setRadius(int r) {
20     return new Circle(this.c, r);
21   }
22
23   public Circle moveTo(Point p) {
24     return new Circle(p.clone(), this.r);
25   }
26
27   public Circle moveTo(int x, int y) {
28     Circle _this = new Circle(c.clone(), r);
29     _this.c.setLocation(x, y);
30     return _this;
31   }
32
33   public Circle moveBy(int dx, int dy) {
34     Point newCenter = new Point(c.x + dx, c.y + dy);
35     Circle _this = moveTo(newCenter);
36     return _this;
37   }
38
39   public Point getLocation() {
40     return c.clone();
41   }
42 }
43
44 class Client {
45   public static void main(String[] args) {
46     Point center = new Point(1,1);
47     Circle circle = new Circle();
48     circle = circle.setRadius(7);
49     circle = circle.moveTo(center);
50     System.out.print("center=" + circle.getLocation() +
51                      ", radius=" + circle.getRadius());
52   }
53 }
```

Figure 1. IMMUTATOR converts a mutable `Circle` (left pane) into an immutable class (right pane).

The reader should also notice that IMMUTATOR avoids naive, excessive cloning. For example, it could have defensively placed all cloning in the constructors. Instead, IMMUTATOR generates a private constructor in lines 10–13 (so that objects cannot enter from client code), and uses cloning judiciously, as needed. Thus it rewrites the mutator method `setRadius` without resorting to cloning. The newly returned circle and the old circle share the same `Point` center. IMMUTATOR clones the center object only when it enters from the client code (lines 23–24), when it escapes (line 40), or when it is transitively mutated (lines 28–29).

Even for this simple example, the reader can see that this refactoring requires interprocedural analysis (e.g., lines 29, 35), which must take pointers into account. Our approach combines the strength of the programmer (the high-level understanding of where immutability should be judiciously employed) and the strengths of the tool (searching through many methods, and making mechanical transformations). IMMUTATOR automatically handles all the rewriting (see Section 4) and the analyses (see Section 5) required to make a class immutable.

3. Immutator

We implemented IMMUTATOR as a refactoring plugin in the Eclipse IDE. The programmer accesses IMMUTATOR from the refactoring menu of Eclipse.

To use IMMUTATOR, the programmer selects a class and then selects the option *Make Immutable*. IMMUTATOR will then present three choices: perform and apply the refactoring (no preview), perform the refactoring and show a preview, or cancel the refactoring.

If the programmer selects to perform the refactoring then IMMUTATOR analyzes and rewrites the code. If the programmer asked for a preview then IMMUTATOR shows the changes in a before-and-after pane (screenshot in Fig 2). In this pane the programmer can further select that only a subset of the changes be applied, or let IMMUTATOR apply all changes.

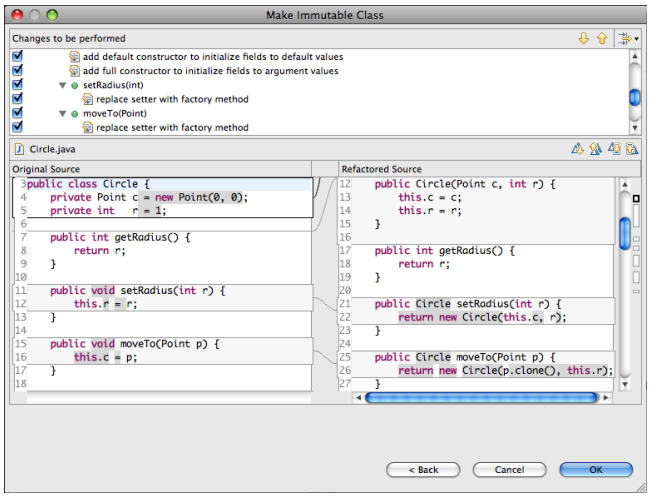

Figure 2. Screenshot of Immutator

Before applying any transformations, IMMUTATOR checks that the input program meets the refactoring preconditions. IMMUTATOR reports to the programmer any preconditions that are not met, and the programmer can decide to ignore the warning and proceed, or can cancel the refactoring, fix the root cause of the warning, and re-run IMMUTATOR.

3.1 Refactoring Preconditions

IMMUTATOR checks several preconditions.

Precondition#1: the target class does not have subclasses. These subclasses could override the target class methods and either expose some state (through objects escaping or entering the state of the target class), or allow mutations to the target class.

Precondition#2: mutator methods in the target class have a `void` return type. Remember that IMMUTATOR converts a mutator method into a factory method that returns a new instance of the target class. Java allows only one value to be returned (Java does not have `out` method parameters). Thus IMMUTATOR could not cleanly rewrite a mutator method into one that returns both a new instance and the old return type. To check this precondition, IMMUTATOR detects all methods that mutate the transitive state of the target class. Section 5.2 presents the details of this analysis.

Precondition#3: the objects entering or escaping the transitive state of the target class implement the `clone` method. Remember that IMMUTATOR clones the entering or escaping objects to prevent client code from mutating the state of the target class directly through these objects. Section 5.3 presents the details of the analysis for finding entering and escaping objects.

Precondition#4: client code does not alias references to instances of the target class. Remember that IMMUTATOR moves the mutation from the object to the reference, by reassigning the reference when the client code invokes a mutator method. IMMUTATOR can only update one reference. Updating aliased references would require deep understanding of the program semantics. Section 5.4.presents the details of the alias analysis.

4. Transformations

This section describes the transformations that IMMUTATOR applies on the target class to guarantee that its state will not change after its creation (see Section 4.1). IMMUTATOR also changes the clients of the target class to ensure that they use the target class in an immutable fashion (see Section 4.1). This is essential to ensure that the client application will not behave differently as a result of the transformations to the target class. We will use the motivating example introduced in Fig. 1 to illustrate the program transformations.

4.1 Target class transformations

Final fields First, IMMUTATOR makes all the fields of the class `final`. The `final` keyword in Java, when applied to fields, forbids field assignments outside of constructors or field initializers. This means that the fields are frozen after construction.

Generate constructors Since final fields can not be modified outside of constructors or field initializers, IMMUTATOR adds two new constructors (see line 5 and 10). We call the first constructor an *empty* constructor since it does not take any arguments. This constructor initializes each field to their initializer value in the original class or to the default value if they had none. The second constructor is a *full* constructor in the sense that it takes one initialization argument for each field. To prevent excessive cloning, IMMUTATOR generated the full constructor as having *private* visibility. If it generated a public full constructor, IMMUTATOR had had to defensively clone all entering object parameters.

Make class final IMMUTATOR also makes the target class final. In Java, the `final` keyword prevents a class from being extended. Therefore, the target can no longer be extended with subclasses containing mutable state.

Convert mutators into factory methods Since the fields are now `final`, methods are forbidden from assigning to them. Furthermore, we consider the state of an objects as its deep, transitive state.

The *transitive state* of an object is the state of the object itself, i.e., its fields, as well as the transitive state of any object that *may* be reached through one of its reference fields.

We call a method a *mutator* method of the target class if the following holds:

- It assigns to a field in the transitive state of an instance of the target class.
- It invokes a method that is a mutator method.

Convert direct mutators One type of mutator methods that are very common in object-oriented programs are methods that assign to a single field. These are often called setter methods and an example is `setRadius` (line 19 in Fig. 1). IMMUTATOR converts a mutator method into a factory method that creates and returns a new object with altered state. Lines 19-21 on the right-hand side of Fig. 1 show the transformation of `setRadius` to a factory method: (i) IMMUTATOR changes the return type to the type of the target class and (ii) it changes the method body to construct and return a new object using the full constructor. The argument to the constructor that sets the `r` field is set to the right-hand-side of the assignment expression. The arguments for the other fields (e.g., `c`) are copied from the current object. This has the effect of creating and returning a new object where the `r` field has the new value, while all other fields remain unchanged.

However, not all mutator methods are simple setters. Some contain many statements while others mutate fields indirectly by calling other mutator methods. The `moveBy` method on lines 33–37 demonstrates both of these traits. It contains two statements, and it mutates `c` indirectly by calling the `moveTo` method.

Lines 33-37 on the right-hand side show how IMMUTATOR transforms `moveBy` into a factory method. It introduces a new local reference, called `_this`, to act as a placeholder for Java's built-in `this` reference. After `_this` is defined (at the statement of first mutation), IMMUTATOR replaces every explicit and implicit `this` with `_this`.

Furthermore, for every statement that calls a mutator method (e.g., `moveTo`), IMMUTATOR assigns the return value of the method

(which is now a factory method) back to `_this`. Thus, the rest of the method sees and operates on the object constructed by the factory method. Finally, the `_this` reference is returned from the method.

An interesting property of this technique is that IMMUTATOR effectively shifts the mutations from the target object to a reference. That is, it shifts the mutations from the object pointed by `this` to the mutation of its reference. Ideally, we would reassign back to `this`. However, in Java, the built-in `this` reference can not be reassigned, therefore IMMUTATOR replaces it with a mutable place-holder, `_this`.

Convert transitive mutators We will present now how IMMUTATOR handles mutations to the transitive state of a target object. Consider the `moveTo(int, int)` method in lines 27–31. Although this method never assigns to the `c` field, it still mutates `c`'s transitive state through the `setLocation` method. Notice that `setLocation` does not belong to the target class (e.g., it belongs to the `java.awt.Point` in the GUI library), therefore IMMUTATOR cannot change `setLocation` into a factory method.

As before, IMMUTATOR creates the `_this` reference, and returns it at the end of the method. Since IMMUTATOR can not rewrite the `setLocation` method, but still must allow the mutation on `c`, IMMUTATOR clones `c` so that the mutation does not affect the original object referenced by `this`. The cloned `c` is passed as an argument to a new `Circle`, assigned to `_this`. Since `_this.c` now refer to a clone of the original `this.c`, we can allow the mutation through the `setLocation`.

Cloning the entering/escaping state Another way how the transitive state of the target object can be mutated is if the client code gets a handle on the internal state, and then mutates it outside of the target class. This can happen in two ways: (i) through objects that are entering in the target object (e.g., `Point p` at line 24) or (ii) through objects that are escaping from the target object (e.g., `c` at line 40).

An object *enters the target class* if it is assigned to a field in the transitive state of the target class, and the object may be visible from client code. For example, the client code holds a reference to the `center` object (line 46), and it passes it on to `moveTo` method in line 49, which then becomes the transitive state of the `Circle`. The client could later mutate `center` directly, thus breaking the encapsulation. If the target class contained explicit constructors with entering objects, IMMUTATOR would have cloned them as well.

We define a *target class escape* as an escape from any of its methods including constructors. An escape from a method means that an object which is transitively reachable from a field of the target class is visible to the client code after the method returns. For example, at line 40, the object pointed by `c` escapes through the return statement, and the client code at line 50 receives it.

If an object enters or escapes then current or future client code may perform any operations on it and we must conservatively assume that it will be mutated.

IMMUTATOR handles entering and escaping objects by inserting a call to the `clone` method to perform a deep copy of the object in question. If the entering or escaping object is itself immutable, IMMUTATOR does not clone it. The current implementation understands as immutable the following classes: `String`, Java primitive wrapper classes (e.g., Integer), classes annotated with `@Immutable`, and classes previously refactored with IMMUTATOR.

When it needs to use a `clone` method that does not exist, IMMUTATOR reports it to the user who needs to correctly implement a deep `clone` method.

4.2 Client transformations

As discussed in the previous section, among other transformations, IMMUTATOR rewrites mutator methods into factory methods. This means that client code that invoked a mutating method now has to invoke a factory method, and must subsequently use the returned new object (with its modified state) to preserve the semantics of the client code.

Consider the client code at the bottom of Figure 1. Upon invoking the mutating methods `setRadius` and `moveTo`, IMMUTATOR reassigns the result of the factory methods back to the `circle` reference.

The insight behind the transformation is that the client code expects a mutation, but in many cases IMMUTATOR can move the mutation to reference *instead* of the object[1].

5. Program Analysis

In the previous section we discussed the transformations to make an existing class immutable. In order to perform these transformations IMMUTATOR performs a number of analyses to establish refactoring preconditions and to collect information for the transformation phase.

At the heart of IMMUTATOR are three analyses. The first two analyses check preconditions and collect information needed to transform the target class. The third analysis checks preconditions and collects information needed to transform the client code.

For the target class analysis, IMMUTATOR detects mutating methods so that it can rewrite them into factory methods. Also IMMUTATOR detects objects that are escaping or entering the target class so that it can clone them.

For the client code analysis, IMMUTATOR detects aliases of target class references on which the client code invokes mutating methods. Such aliases would cause the client code transformation to break the program semantics and must therefore be flagged to the programmer.

5.1 Analysis Data Structures

IMMUTATOR creates several data structures and uses them in the program analyses. Keep in mind that IMMUTATOR does not perform a whole-program analysis, but *on-demand* analysis. That is, IMMUTATOR analyzes only the code of the target class, the code invoked from the target class, as well as the client code that uses the target class. A client can only call non-private methods from the target class. These are the *API* methods of the target class.

The first data structure is a *callgraph* (CG) starting from every API method of the target class. This callgraph is used to find mutators as well as entering and escaping objects.

For each node in the callgraph IMMUTATOR also constructs a *control flow graph* (CFG) that is used later to find transitive mutations and to perform a liveness analysis required by the alias analysis. We construct both of these data structures using the WALA analysis library [1].

In addition to these control-flow structures IMMUTATOR also builds a *points-to graph* (PTG). Points-to analysis establishes which pointers (or *references* in Java terminology) point to which *storage locations*. We model the heap storage locations as object allocation sites.

The points-to graph that IMMUTATOR creates is illustrated using a simple client program in Fig. 3. The graph contains two types of nodes: references, depicted graphically as ellipses, and heap-allocated objects depicted graphically as rectangles. The formal arguments of a method are placed on the border of its bounding box. Directed edges connect references to the objects they point to. For example, the object allocated on line 2 is represented by the rectangle `Circle:2` and the reference it is assigned to on the same line is represented by the `circle1` ellipse. This object has one field `c`, which is allocated in the field initializer of class `Circle`. Fields are also connected to their objects by directed edges. The points-to

[1] See section 5.4 for a discussion on when this is not sufficient and how such cases can be detected

```
1    public void client() {
2      Circle circle1 = new Circle();
3      Circle circle2 = circle1;
4
5      Point center = new Point(0, 0);
6      circle2.moveTo(center);
7    }

8    public void moveBy(int dx, int dy) {
9      Point newCenter = new Point(c.x + dx, c.y + dy);
10     moveTo(newCenter);
11   }
```

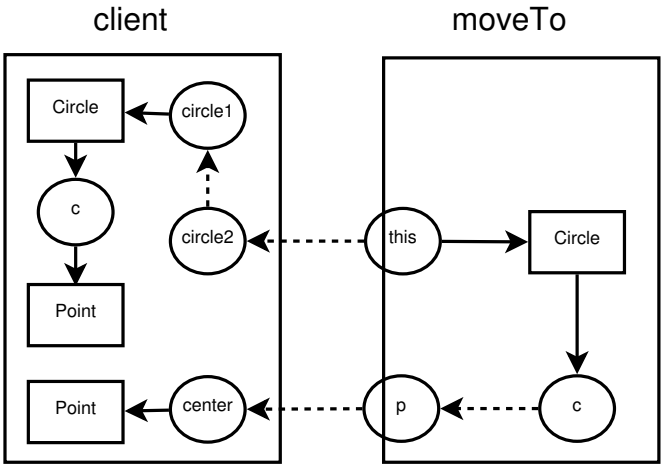

Figure 3. An example of a points-to graph

graph only captures relations between objects and references and does not include scalars/primitives.

Notice that the assignment on line 3 creates an alias between the references `circle2` and `circle1`. This is represented in the points-to graph as a dashed arrow and we call this a *deferred edge*. This deferred edge means that `circle2` can point to any objects that `circle1` is pointing to. We also use deferred edges to represent the relations between formal and actual arguments since Java is a pass-by-value language where actuals are copied into the formals.

IMMUTATOR constructs the points-to graph using an inclusion-based (Andersen-style [2]) points-to analysis. The analysis is *flow-insensitive* (it does not take into account the order of statements) and *context-insensitive* (it does not take the calling context into account).

Now we explain the interprocedural part of the points-to analysis. First, the IMMUTATOR computes for each method a summary of its points-to relations. In a second pass, it traverses all the methods in the reverse topological order in which they appear in the call graph (i.e., from the leaf methods up to the entry methods). For each method `m`, it goes through the call sites, and connects `m`'s points-to graph to the callees' points-to graph. When doing this, it adds deferred edges from the formal to the actual method parameters (including the receiver of the method).

Note that IMMUTATOR constructs additional nodes that do not exist in the program if they are needed to complete a method summary. One such example is the `Circle` allocation site and its `c` field in the `moveTo` method. When IMMUTATOR creates the summary for `moveTo` its `this` reference is not connected to any allocation sites. In order to add the deferred edge that represents the assignment of `p` to `c` IMMUTATOR therefore constructs additional nodes as needed.

For example, the statement on line 6 on the left-hand side of Fig. 3 invokes the `moveTo` method (whose implementation is shown on line 8). The right hand-side of Fig. 3 shows the points-to summary of the `moveTo` method and the client method, with deferred edges connecting the formal argument (`p`) to the actual argument (`center`), and the `this` reference of `moveTo` to the actual receiver (`circle2`).

5.2 Detecting Transitive Mutators

The goal of this analysis is to find all methods that are mutating any part of the transitive state of the target object either directly or indirectly by calling another mutator method.

Fig. 4 shows the pseudocode of the algorithm for detecting mutator methods. The algorithm takes as input the set `M` of methods

```
INPUT: M = Set of Methods in CG, MTC = Set of Methods in Target Class,
       PTG = Points-to Graph,
OUTPUT: Mut = Set of mutator methods

Mut = ∅

// Step 1: Find the transitive state of the target class
TargetNodes = ∪_{m∈MTC}(transitiveClosure(this))

// Step 2: find transitive mutators
forEach m in M
  forEach fieldAssignment:< o.f = expr >
    if o reachesThroughDeferredEdge TargetNodes
      Mut = Mut ∪ m

// Step 3: find indirect mutators
forEach m in M, post-order
  forEach m' in calleesOf(m)
    if m' ∈ Mut
      Mut = Mut ∪ m
```

Figure 4. Pseudocode for detecting transitive and indirect mutating methods

in the call graph, the set `MT` of methods declared in the target class, and the interprocedural points-to graph presented in Section 5.1. The output of the algorithm is a set `Mut` of mutator methods.

In the first step, the algorithm finds the nodes representing the transitive state of the target class. To do so, the algorithm computes the transitive closure of the `this` reference to the target class, i.e, all nodes in the points-to graph reachable from `this`. These nodes, called `TargetNodes` are the set union of all nodes reachable from `this` in target class methods.

In the second step, the algorithm finds all the transitive mutating methods. The analysis visits all field assignment instructions in the target class methods and in methods invoked from the target class. For each assignment it checks whether the receiver of assignment is a node that can reach one of the nodes in the transitive state of the target class, following deferred edges. If it can, this means that the instruction assigns to the transitive state of the target class, and in this case the algorithm marks the method as a direct mutator.

In the third step, the algorithm propagates the mutation information from direct mutators across the call graph. To do this, it visits in a post-order fashion (i.e., reversed topological order) the methods that appear in the call graph and propagates the mutation information from the leaf nodes of the call graph up to the entry

```
INPUT: PTG = Points-to Graph,
       API = Set of API Methods from Target Class
OUTPUT:
      Entering = Set of Entering Objects
      Escaping = Set of Escaping Objects

Entering, Escaping = ∅

// Step 1: Find the transitive state of the target class
TargetNodes = ∪_{m∈API}(transitiveClosure(this))

// Step 2: Find the transitive closure of the boundary nodes
OutsideNodes = ∪_{m∈API}(transitiveClosure(actuals) ∪
                    transitiveClosure(returns) ∪
                    transitiveClosure(statics))

// Step 3: Find the escaping objects
forEach deferredEdge e in PTG
  if (e.source ∈ OutsideNodes) && (e.sink ∈ TargetNodes)
    Escapes = e.sink ∪ Escapes

// Step 4: Find the entering objects
forEach deferredEdge e in PTG
  if (e.source ∈ TargetNodes) && (e.sink ∈ OutsideNodes)
    Enters = e.source ∪ Enters

// Step 5: Clone entering and escaping objects
forEach n in (Enters ∪ Escapes)
  clone (n)
```

Figure 5. Pseudocode for detecting entering and escaping objects

methods. If method `m` calls `m'`, and `m'` is a mutator method, then `m` becomes a mutator method.

Since this step is performed post-order we only have to do one pass over the callgraph. The reason why we only need one pass, even in the case where there are loops in the call graphs caused by recursion, is that there are no kill sets. Once a method is marked as a mutator there is no instruction that can un-mark it.

5.3 Detecting Escaping/Entering Objects

The goal of this analysis is to find entering or escaping objects to/from the target class. These objects allow a client to get a handle on the internal state of the target class, potentially mutating the target class directly through these objects. To prevent such mutations, once the analysis found these objects, it creates clones that are passed into the target class, or returned from the class.

We care only about escaping objects that refer to mutable transitive state of the target class (i.e., `TargetNodes`) and entering objects that have mutable transitive state that the target class sets a reference to. An escaping object that has no transitive mutable state, or an entering object that does not establish a new reference in the target method do not concern us. This is a fundamental difference from the previous work in escape analysis [7, 29].

Fig. 5 shows the pseudocode of the algorithm for detecting entering or escaping objects. The algorithm takes as input the points-to graph presented in Section 5.1. The output of the algorithm are two sets, `Entering` containing objects that enter the target class, and `Escaping` containing objects that escape the target class.

In Step 1, the algorithm labels the nodes that form the transitive state of the target class. The transitive state, denoted by the `TargetNodes` set, is the transitive closure of the `this` target reference.

In Step 2, the algorithm labels the nodes that are outside of the target class and are interfacing with the target class. These are nodes through which a client code interacts with the target class, thus these are the nodes through which objects can enter or escape. We call these nodes *boundary nodes*: they are at the boundary with the target class.

```
1   public Point getLocation() {
2     return c;
3   }
```

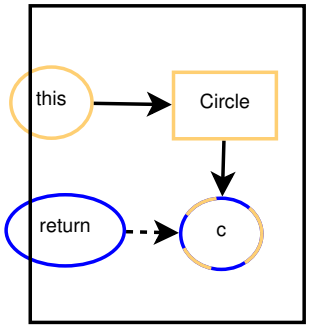

Figure 6. Example of an escaping object

The boundary nodes are:

- actual arguments passed to API methods
- objects returned from API methods
- statically allocated objects (i.e., stored in static fields). These can cross any class boundaries.

The algorithm computes the transitive closure of the boundary nodes, and labels that set `OutsideNodes`[2].

In Step 3, the algorithm finds the escaping objects. Intuitively, if we imagined the target class being a black-box, the escaping objects are those target objects that might be *seen* from the outside world, and are in the transitive state of the target class. To find these objects, the algorithm visits all the deferred edges which start in `OutsideNodes` and end in the `TargetNodes` (since we only care about escaping objects that refer to mutable transitive state of the target class). For such an edge, the algorithm adds the sink target node to `Escapes`.

In Step 4, the algorithm finds the entering objects. Intuitively, if we now imagined the outside world being a black-box, the entering objects are those outside objects that can be *seen* from the target class. To find these objects, the algorithm visits all the deferred edges which start in the `TargetNodes` and end in the `OutsideNodes` (since we only care about entering objects that have mutable transitive state that the target class sets a reference to). For such an edge, the algorithm adds the incoming outside node to `Enters`.

In Step 5, the algorithm *clones* the entering or escaping objects, to break the direct connection between the outside and the target class. If the entering or escaping objects were immutable, IMMUTATOR would not clone them.

Fig. 6 shows a concrete example of an escaping object. The figure shows the points-to graph for the `getLocation` method, with an additional node representing the return. We color the transitive state of the target object (which is the transitive closure of `this`) with orange. We color the outside nodes with blue. In this example, the only boundary node is the return node, and its transitive closure includes `c`. Notice that `c` is colored with both blue and orange. In this case, `c` escapes because it can be seen from the outside (it's blue), and it is part of the transitive state of the class (it is orange). Therefore, IMMUTATOR clones it.

Fig. 7 shows a concrete example of an entering object. The figure shows the points-to graph for the `moveTo` method. The transitive state of the target class is colored orange, and the transitive closure

[2] notice that these nodes are a subset of all the nodes outside of the target class

```
1   public void moveTo(Point p) {
2     this.c = p;
3   }
```

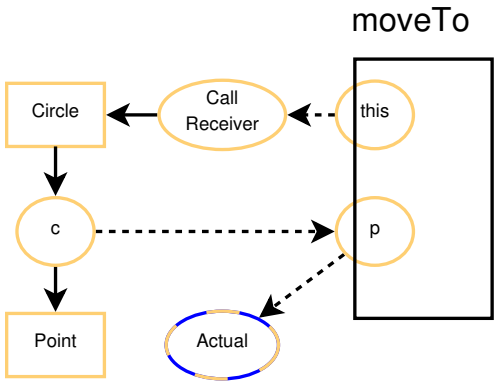

Figure 7. Example of an entering object

```
1  Circle c1 = new Circle();
2  Circle c2 = c1;
3
4  c2.setRadius(7);
5
6  System.out.println(c1.getRadius());
```

Figure 8. Simple Alias Example

of the boundary nodes (i.e., the first actual argument) are marked blue. The reference `p` is part of the transitive state (it is marked orange), and it can point to objects outside of `moveTo` (i.e., the actual argument).

For pedagogical reasons, we chose to show simple examples of entering or escaping objects. In the codes illustrated in Fig. 7 and 6, it is very easy to spot the entering/escaping objects. However, in many cases, it is more difficult to notice, especially, if the object enters or escapes through a container, or a method call. Section 7.3 shows a concrete example of a state object escaping through an iterator container. The open-source developer overlooked such an escaping object, but IMMUTATOR correctly finds it.

5.4 Detecting Aliased References in Client Code

IMMUTATOR rewrites the client code to use the target class in an immutable fashion by assigning the new object returned from a factory method back to the receiver of the call. In our motivating example (Fig. 1) we show the case when such rewriting preserves the behavior of the client code.

However, it is not always possible to preserve the client code semantics via a reference reassignment. Fig. 8 shows such an example. Here `c1` and `c2` are aliased. In the original code, line 6 prints value 7. If we rewrite line 4 to assign the result back to `c2`, then `c1` would point to the original object, and line 6 would print 1.

In the original program, when invoking a mutator method, this mutates the state of the object. IMMUTATOR shifts this mutation from the object to the receiver reference of the mutating method by assigning the object returned from the factory method back to the receiver.

IMMUTATOR can only update a single reference[3]. Thus, it must ensure that there are no other references that may point to the same

[3] updating more than a single reference requires deep understanding of the semantics of the code

```
1  public void printCircle() {
2    Circle c1 = new Circle();
3
4    initRadius(c1);
5
6    System.out.println(c1.getRadius());
7  }
8
9  public void initRadius(Circle c2) {
10   c2.setRadius(7);
11 }
```

Figure 9. Inter-procedural Alias Example

object. Otherwise, the other references would not be mutated and would not see the new state as illustrated by the example in Fig. 8.

The goal of this analysis is to find out whether the receiver of a mutating method is the *only reference* to the object that will be mutated. If there exist other aliased references, IMMUTATOR reports them to the programmer who can intelligently break the alias and rerun the tool. The programmer is the right person to make these changes since they require understanding the semantics of the code at a level deeper than any tool can understand.

The analysis detects whether the receiver of a call to a mutating method has any *live aliases* at the call site. We designed an inter-procedural analysis, but we will first explain the intra-procedural part of the analysis.

In order to find the live aliases of the receiver at the mutating call site, IMMUTATOR first finds the live references at that program line. A reference is live at a line if it is defined before that line and used at least once after it. Otherwise it is dead. IMMUTATOR detects live references using a standard liveness analysis that propagates liveness information backwards through the CFG.

The points-to set (*pt*) of a reference is the set of allocation sites that the reference may point to. At each call site of a mutating method, IMMUTATOR computes the points-to set of the receiver (pt(receiver)) and the points-to set of the live references ($pt(LiveSet)$) at that program line. If the intersection between them is not empty, then IMMUTATOR reports an alias. This condition is shown in the following formula:

$$pt(receiver) \cap pt(LiveSet - receiver) \neq \emptyset$$

IMMUTATOR removes the receiver from the $LiveSet$ because we are not interested in aliases with itself.

Note that IMMUTATOR's alias detection is conservative: it is a *may* alias analysis. Its precision depends mostly on the precision of the points-to analysis. To improve precision, IMMUTATOR does not use the previously computed points-to graph (Andersen-style), but it uses a more precise and fast, context-sensitive, on-demand pointer analysis [25].

Interprocedural alias detection Now we explain how IMMUTATOR detects live aliases inter-procedurally. Consider the example client methods in Fig. 9, which is similar to the previous example (Fig. 8). The difference is that `c2` and the call to the mutator method (`setRadius`) are now in a different method. However, this does not change the fact that `c2` aliases `c1` since at the invocation, `c1` is copied into `c2` due to Java's *pass-by-value* calling semantics.

In the original code, line 6 would print value 7, since the object's state is mutated inside the `initRadius`. If we rewrote line 10 to assign the newly constructed object back to the receiver (`c2`), then the resulting code would print 1. This is because `c1` would point to the original object, not to the newly constructed object, thus it would not see the new state.

IMMUTATOR performs an inter-procedural analysis to detect aliases like `c1` in Fig. 9. It first visits all methods in the client code that call one of the mutating methods in the target class. For each client

```
1  Circle circles[] = new Circle[10];
2
3  public void printPoint() {
4    Circle c1 = new Circle();
5    circles[0] = c1;
6
7    c1.setRadius(7);
8
9    System.out.println(circles[0].getRadius());
10 }
```

Figure 10. Heap Alias Example

method, IMMUTATOR adds the receiver of each mutating call to the set $MutateSet$ of that client method.

The $MutateSet$ of a method contains all references (both the ones that appear directly in that method, and the ones that appear in callees) that will be mutated due to factory methods.

IMMUTATOR then performs a post-order traversal of the client code's callgraph to propagate `MutateSets` upwards. For each method `m`, the $MutateSet$s of its callees are added to its $MutateSet$. Intuitively, these references may be mutated as the result of a call to `m`.

Once the `MutateSet` of each method has been collected, IMMUTATOR performs a live variable analysis like in the intra-procedural case. When the live variable analysis reaches any call site, IMMUTATOR computes the intersection between the points-to set of all of the references in the `MutateSet` with the points-to set of the references in the $LiveSet$:

$$pt(MutateSet) \cap pt(LiveSet - MutateSet) \neq \emptyset$$

As before, if the intersection is not empty then IMMUTATOR reports an alias.

We illustrate the analysis using the code example from Fig. 9. The $MutateSet$ of method `initRadius` contains `c2`, the receiver of a mutating call. The analysis propagates this $MutateSet$ upwards, to the caller `printCircle`. Then IMMUTATOR computes the liveness information in `printCircle`. When it reaches the call to `initRadius`, it intersects the points-to set of the references in the $LiveSet$ (i.e., `c1`) at this line, with the points-to set of the `MutateSet` (i.e., `c2`). The intersection contains the allocation site at line 2, thus IMMUTATOR reports an alias to the programmer.

Heap alias detection Another way how objects can be aliased is through the heap. Fig. 10 shows how a reference can be copied into a location on the heap. Line 4 creates a new circle and assigns it to `c1`. On line 5, `c1` is assigned to the first slot of the `circles` array. Then, on line 7, the `setRadius` method is invoked on `c1` setting the circle object's radius to 7. Finally, line 9 prints value 7 by invoking `getRadius` on the reference in the first array slot (which is aliased to `c1`).

If IMMUTATOR would rewrite line 7 to assign the returning object back to the receiver, then line 9 would print 1 instead of 7 (because the first slot of the array would still point to the old object). IMMUTATOR handles such cases by adding every reference that is assigned into a heap location to a set of global live variables, called $gLiveSet$. When the analysis reaches a method that causes a reference to be mutated (has a non-empty $MutateSet$), IMMUTATOR compares both the points-to set of $LiveSet$ and the points-to set of $gLiveSet$ to the points-to set of $MutateSet$. As before, if the intersection is not empty then IMMUTATOR reports an alias:

$$pt(MutateSet) \cap pt((LiveSet \cup gLiveSet) - MutateSet) \neq \emptyset$$

6. Discussion

Currently, IMMUTATOR handles most of the complexities of an OO language like Java: arrays, aliases, polymorphic methods. It models arrays as an allocation side with just one field, which is the array contents. Although this abstraction does not allow IMMUTATOR to distinguish between array elements, it is sufficient for detecting objects entering or escaping through array, as well as assignments to array elements. IMMUTATOR disambiguates polymorphic method calls by computing the concrete type of a receiver using the results of the points-to analysis explained in Section 5.1. We left for future work handling of inner classes.

Since our refactoring for immutability changes the signature of mutating methods, it changes the API of a target class in a manner that is not backwards-compatible. That is, if the target class is a library class, client code that uses the target class will no longer work correctly with the refactored class.

IMMUTATOR can only update client source code that it has access to. For updating the source code of remote clients, programmers can use one of our previously developed solutions [8], *record-and-replay* of refactorings. This technique is also incorporated in the official release of the Eclipse IDE. The enhanced refactoring engine automatically *records* information about the MakeImmutable refactoring into a refactoring log. The library developer can ship this log, and a client developer loads it into his refactoring engine and `replays` the MakeImmutable refactoring. Now that IMMUTATOR has access to the client code, it can correctly refactor it.

Currently we are also investigating another solution which does not require any changes to the client code. In the future, IMMUTATOR could refactor a mutable class into two classes: an immutable class (as it does currently), and a mutable wrapper class that encapsulates the immutable class as a *delegate* field. The wrapper implements the same method protocol as the original class, but delegates all the method calls to the immutable class. Upon invoking a mutator method, the wrapper class invokes the corresponding factory method from the delegate, and reassigns its delegate field to the object returned by the factory method. Using this solution, the reference reassigning happens inside the wrapper class, without requiring any changes on the client code.

Limitation Since IMMUTATOR analyzes bytecodes, it correctly handles calls to library methods. However, if the program invokes native code, IMMUTATOR can not analyze it. Also, like any other practical refactoring tool, ours does not handle uses of dynamic class loaders and reflection. The current implementation does not handle inner classes, which precludes IMMUTATOR from running on JDigraph case study (see Section 7.3. However, we plan to support inner classes in the near future. This requires more engineering, but no fundamental changes to the algorithm.

7. Evaluation

7.1 Research Questions

In this section we answer the following research questions:

- Is IMMUTATOR useful?
- Does IMMUTATOR save programmer's time?
- Is refactoring with IMMUTATOR safer than refactoring manually?

To answer these questions we need two versions of the same class: one mutable and one immutable. We use IMMUTATOR to convert the mutable class into an immutable class, and then compare the manual vs. automatically refactored code.

Although it is relatively easy to find immutable classes in existing code-bases, it is much harder to find their mutable counterparts. This happens because (i) developers never wrote a mutable counter-part, or (ii) they no longer maintain the mutable version of the class, or (iii) they still maintain the mutable counter-part, but is located in a different module and has a different name.

	Mean	Std.Dev.	Min	Max
Years Programming	12.3	4.2	7	20
Years Java Programming	7.1	2.5	5	10
Years Using Eclipse	4.1	1.9	1	6

Table 1. Demographics of the six participants in the experiment.

We approach the problem of finding pairs of mutable/immutable classes from two sides. We designed an "in the lab" controlled experiment where programmers start from a mutable class and create an immutable counter-part. We also conduct "in the wild" case studies where we start from immutable classes in open-source projects and trace back the mutable counter-part.

The two complementary empirical methods strengthen each other: the controlled experiment allows us to better quantify the programmer's time to refactor, where as the case studies give more confidence that the lab findings generalize to real-world situations.

7.2 Controlled Experiment

7.2.1 Experiment's Design

We asked 6 programmers to manually refactor for immutability 9 classes from the JHotDraw framework [17]. JHotDraw is an open-source 2D graphics framework for structured drawing editors, written in Java. It is based on Erich Gamma's original JHotDraw, thus it is considered by many to be a high-quality design project.

We gave each programmer a 1-hour tutorial on making classes immutable, and then we asked them to refactor one or two JHotDraw classes and to report the refactoring time. We also used IMMUTATOR to refactor the same classes, and we compared the results of IMMUTATOR with the results of the programmers.

7.2.2 Demographics

We asked 6 graduate students at UIUC to volunteer in a software engineering controlled experiment. We were looking for students that had lots of programming experience and were comfortable with navigating through open-source code using the Eclipse IDE. Table 1 shows the demographics of the participants in the experiment. Although these are all graduated students, they are expert programmers, with several years of programming experience.

We asked the subjects not to refactor for more than one hour. The subjects did not know the other participants, nor the purpose of the experiment. We wanted them to refactor for immutability, as if they were doing this task as part of their daily programming.

7.2.3 Tasks

Each subject received an Eclipse project pre-loaded with the JHotDraw project. We asked them to manually refactor one or two specific classes from the JHotDraw framework. We chose mostly classes from JHotDraw's `Figure` class hierarchy. These are classes that made sense to become immutable. Table 2 lists these classes. We split the 9 JHotDraw classes among participants so that none of them got a class larger than 400 LOC: we gave two classes per participant if the classes were smaller than 200 LOC, or one class per participant if the class was between 300–400 LOC.

The participants did not have to use the same techniques as in the tutorial, but we asked them to convert mutator methods into factory methods, instead of leaving the methods blank or throwing an exception.

Since the `Figure` classes are part of a larger class inheritance hierarchy, we knew that refactoring the target class might require changing many other classes in the class hierarchy (for example, when turning an overridden mutator method like `moveBy` into a factory method). We told the participants to change only the target class. That is, treat the target class as if it was the only class in the hierarchy. This avoided having them spend lots of time repeating the same change across a whole hierarchy of overridden methods.

7.2.4 Variables:

Controlled Variables. All subjects used Eclipse, Java, and JHotDraw. All subjects were exposed to an extensive tutorial explaining the mechanics and analyses of the refactoring (e.g., transitive mutator methods, entering/escaping objects, etc.).

Independent Variables. Refactoring for immutability by hand, versus refactoring with IMMUTATOR.

Dependent Variables. Time spent to refactor, the number of errors: failures to prevent entering/escaping objects into/from the transitive state of the target class, failures to prune the target class of transitive mutating methods.

7.2.5 Experimental Treatment

Now that we had pairs of mutable/immutable classes manually refactored by programmers, we ran IMMUTATOR on the mutable classes, and compared the manually refactored and IMMUTATOR-refactored against a golden standard. We obtained the golden standard by refactoring the same classes ourselves and carefully examining all the transitive mutators, and entering/escaping objects.

Table 2 lists the results of refactoring manually vs. refactoring with IMMUTATOR. The results show that refactoring with IMMUTATOR is *faster*: to refactor the 9 JHotDraw class, the participants took 220 minutes, whereas IMMUTATOR took 1.5 minutes. Also, refactoring with IMMUTATOR is safer: whereas the participants made 51 errors, IMMUTATOR made 27 errors. We inspected IMMUTATOR's errors and they are all due to bugs in the current implementation. We also list in the "Other" column how many times the programmers forgot to add `final` keywords. Also the programmers used 6 excessive cloning compared to IMMUTATOR.

7.2.6 Threats to Validity

Construct Validity One could ask why we compare the manually-refactored and IMMUTATOR-refactored outputs in terms of time and refactoring errors. We believe that a software tool should improve programmer's productivity (thus we measure refactoring time) and software's quality (thus we measure the rate of errors).

One could also ask why we used IMMUTATOR ourselves rather than using two groups of programmers: one controlled group who refactored manually, and one group who refactored with IMMUTATOR. Due to the high-level of automation in this refactoring, the programmers only need to select the target class (since they know which classes make sense to become immutable). The programmers need to intervene only if the refactoring preconditions were not met. Since we asked the programmers to treat each `Figure` class in isolation, all the refactoring preconditions were met.

Internal Validity One could ask whether the design and the results of the experiment truly represent a cause-and-effect relationship. For example, maybe the participants made errors or took a long time to refactor because they were not familiar with the JHotDraw code. Maybe the participants would have been more productive with refactoring their own code. First, JHotDraw is a well-documented, high-quality open-source project. Second, the `Figure` classes all implement the protocol that one encounters when programming with *standard* graphical toolkits: methods like `moveBy` to move a figure, `draw` on a graphical context, `read` and `write` from/to input/output streams. Third, by giving the participants classes from the same code-base that none of them developed, we could control many experiment variables (the level of code complexity in the target class, the learning effects, i.e., how recently has the programmer worked with a target class, etc.).

External Validity One could ask whether the results are applicable and generalizable to other software projects. We only used

Class	SLOC	Programmer Time [min]	IMMUTATOR Time [sec]	Transformations			Programmer Errors			
				Entering	Escaping	Mutators	Entering	Escaping	Mutator	Other
EllipseFigure	104	17	7	4	0	4	5	0	1	1
ArrowTip	145	15	5	0	0	4	0	0	0	0
ColorEntry	97	16	3	1	0	0	1	0	0	1
ImageFigure	154	20	7	1	0	3	4	0	2	0
LineConnection	344	53	14	4	2	7	2	1	2	0
FigureAttributes	204	24	12	1	1	2	1	1	1	2
TextFigure	381	45	25	3	3	16	6	2	7	0
PertFigure	311	30	20	5	0	11	5	0	10	1
Total	1740	220	93	19	6	47	24	4	23	5

Table 2. Results of the controlled experiment using classes from JHotDraw 5.3. The table shows the mutable classes, their size in non-blank, non-comment LOC as reported by SLOC, the programmer's time to refactor the class into an immutable class, and the IMMUTATOR's time. Next columns show the number of program elements that had to be refactored: the number of entering/escaping objects that had to be cloned, the number of mutator methods that had to be converted into factory methods. The last columns show the programmer's errors: the number of entering or escaping objects that can reach the client code, and the number of mutator methods that programmer forgot to handle.

one single project, JHotDraw, and one single population, graduate students from UIUC. Maybe the results are not applicable to other projects, programming languages, or programmers.

All the subjects of our experiment are graduate students. This means that we had a relatively homogeneous pool of participants, so we expected little variation between their performance. On the other hand, maybe full-time programmers made fewer errors when refactoring for immutability. However, as one can see from the demographics data in Table 1, most of the participants had extensive programming experience, and this was not the first time when they encountered immutability.

In the open-source case studies described in Section 7.3, the data shows that even full-time programmers who are very familiar with a wider range of projects still make errors.

Reliability The JHotDraw classes can be found online [17] and the output of refactorings can be found online [14], so our results can be replicated. We also plan to release IMMUTATOR as open-source, once we get approval from our funding agencies.

7.3 Case Studies

We also conducted case studies of how open-source programmer refactored for immutability. We started from immutable classes, and traced back the mutable counter-part.

7.3.1 Experimental Setup

To find existing immutable classes in real-world projects we used a code search engine (krugle[4] or google code search) and searched for Java classes whose name contains the word 'Immutable'. These are classes that are very likely to be immutable. The documentation of these classes confirms that developers intended those classes to be immutable. We use one more heuristic: some open-source projects have the convention that all immutable classes implement a tag interface called `Immutable`. Thus, we also searched for classes implementing `Immutable`.

To find mutable counter-parts, we used three techniques:

1. Mining version control systems. Starting from an immutable class (say $\texttt{A}_{Immut}$), we searched back in the version control history until we found a version of the class ($\texttt{A}_{Mut}$) that is mutable.
2. Studying parallel class hierarchies. Sometimes, programmers support two versions of the same class, in two parallel class hierarchies. One class hierarchy contains mutable classes, say class $\texttt{A}_{Mut}$. The other class hierarchy contains immutable classes, say $\texttt{A}_{Immut}$ which mirror their mutable counterparts.
3. Roundtrip: start from immutable classes, then make mutable by hand. We started from an existing immutable class, $\texttt{A}_{Immut}$, and we manually made some changes into the code so that the class becomes mutable ($\texttt{A}_{Mut}$).

Once we have a pair $\langle \texttt{A}_{Mut}, \texttt{A}_{Immut} \rangle$, we use IMMUTATOR to refactor $\texttt{A}_{Mut}$ into a class $\texttt{A}_{ImmutWithTool}$, and then compare the class refactored manually by open-source developers ($\texttt{A}_{Immut}$) with the output of IMMUTATOR, $\texttt{A}_{ImmutWithTool}$.

We checked carefully that the output of IMMUTATOR was correct, i.e., IMMUTATOR identified and refactored correctly all entering, escaping, and mutator methods. The comparison with $\texttt{A}_{Immut}$ revealed errors that the open-source programmers made: they forgot to clone some entering or escaping objects, or they left in the immutable class some methods that can still mutate the state.

Ideally, we would have recorded the time that it took the open-source programmers to refactor a mutable into an immutable class. However, we do not have access to those developers, and it would be very unlikely that they remembered precisely how long it took them to perform a refactoring. Therefore, we measured how long it took us to analyze the safety of the refactoring. That is, we took an immutable class, and we carefully examined the class to detect all entering, escaping, and mutating methods (this time only includes analyzing the target class, not the client code). Since we are now experts at identifying all the problems involved when performing this refactoring, we expect that other developers would use at least the same amount of time.

7.3.2 Results

Table 3 presents the results. First column lists the open-source project from which we gathered the immutable classes. The next column shows the immutable classes that we took into consideration from each project. Next we show the size of each class, in non-blank, non-comment lines of code (SLOC). The next column shows the time it took us to manually perform the safety analysis. Next column shows the time it took IMMUTATOR to analyze.

The next set of three columns shows how many of the artefacts in the mutable class had to be refactored: the number of entering or escaping objects that have to be cloned, and the number of mutating methods that have to be converted into factory methods.

The last set of three columns shows the errors in the immutable classes: how many entering or escaping objects the open-source programmers forgot to clone, and how many mutating methods they still left in the immutable class. We were not able to run IMMUTATOR

[4] http://www.krugle.org/kse/entcodespaces/DyhKoF

Project	Class	SLOC	Manual	Tool	Transformations			Programmer Errors		
	Immutable/Mutable		Analysis Time	Analysis Time	Entering	Escaping	Mutators	Entering	Escaping	Mutator
JDigraph	ImmutableBag/MapBag	361	15 min	NA	1	-	-	1	-	-
	FastNodeDigraph/AbstractFastNodeDigraph	25	5 min	NA	2	-	-	2	-	-
	HashDigraph	25	5 min	NA	2	2	-	2	-	-
	ArrayGrid2D	160	20 min	NA	2	2	-	2	-	-
	MapGrid2D	160	20 min	NA	2	2	-	2	-	-
WALA 1.3	ImmutableByteArray	76	5 min	6 sec	2	-	-	1	-	-
	ImmutableStack	213	10 min	3 sec	3	3	4	2	3	-
java.util.Collections 1.5.0	SingletonMap.ImmutableEntry/HashMap.Entry	33	1 min	1sec	2	2	-	2	2	-

Table 3. Case studies of using IMMUTATOR on different open-source projects.

on the JDiGraph case study because it currently does not handle inner classes, and all those classes contained several inner classes.

We illustrate one of the programmer's errors in `ImmutableStack` from the WALA open-source project. This immutable class lets some internal state stored in the `entries` field escape. At first sight, it is not obvious, since no fields are directly returned. However, `entries` is passed to an `ArrayIterator` that *is* returned, so `entries` escapes too:

```
public Iterator<T> iterator() {
  if (entries.length == 0) {
    return EmptyIterator.instance();
  }
  return new ArrayIterator<T>(entries);
}
```

Some client code can now use the returned iterator to fetch any element of `entries` and mutate it directly.

8. Related Work

Specifying and checking immutability There is a large body of work in the area of *specifying* and *checking* immutability [5, 20, 28, 32].

Pechtchanski and Sarkar [20] present a framework for specifying a richer set of immutability constraints along three dimensions: lifetime, reachability, and context. The lifetime specifies the duration of immutability, e.g., the whole lifetime of an object, or only during a method call. The reachability specifies what subset of the transitive state is immutable, e.g., shallow or deep immutability. The context specifies whether immutability is enforced only during the context of one method, or all methods. IMMUTATOR enforces deep immutability for the whole lifetime of an object, on all method contexts.

Birka, Tschantz, and Ernst [5, 28] present Javari, a type-system extension to Java for specifying *reference immutability*: an object can not be mutated through a particular reference, though the object could be mutated through other references. In contrast, *object immutability* specifies that an object can not be mutated through any reference. Reference immutability is more flexible, but weaker than object immutability. IMMUTATOR enforces object immutability.

Zibin et al. [32] build upon the Javari work and present IGJ that allows specifying both reference and object immutability. Whereas Javari requires extending the Java language with two keywords, `readonly` and `mutable`, IGJ specifies transitive immutability using the Java generics mechanism (no language extensions).

These systems are very useful to document the intended usage and to detect violations of the immutability constraints. But they leave to the programmer the tedious task of removing the mutable access. In addition to specifying and checking immutability of the target class, IMMUTATOR also performs the tedious task of getting rid of mutable access. IMMUTATOR rewrites the target class (converts mutators into factory method, clones the state that would otherwise be mutated) and the client code.

Supporting program analyses Many of the components of our program analyses have previously been published: detecting side-effect free methods [3, 22–24], escape analysis [7, 29], and alias analysis [31]. However, we have synthesized these pieces in a novel way: we are presenting the first use of such analyses for refactoring to immutability.

Side-effect analysis [3, 22–24] uses interprocedural alias analysis and dataflow propagation algorithms to compute the side effects of functions. There are two major differences between these algorithms and IMMUTATOR's analysis for detecting mutator methods. First, the search scope is different. We are only interested to side-effects to variables that are part of the transitive state of the target class, whereas previous work determines all side-effects (including side effects to method arguments that do not belong to the transitive state). Consider method `drawFrame` from `TextFigure` in JHotDraw (this method is implemented by all other `Figure` subclasses):

```
public void drawFrame(Graphics g) {
  g.setFont(fFont);
  g.setColor((Color) getAttribute("TextColor"));
  g.drawString(fText, ...);
}
```

The previous algorithms would determine that `drawFrame` is a mutator method, because it has side effects on the graphics device argument, `g`. However, if we refactored `TextFigure`, `drawFrame` would not mutate the transitive state of the target class, thus IMMUTATOR does not clone the graphics device.

Second, we are interested in distinguishing between (i) methods in the target class that directly or indirectly assign to the fields of the target class and (ii) methods outside the target class (potentially in libraries) that do not assign to target class' fields, but mutate transitively these fields. IMMUTATOR converts the former mutators into factory methods, and rewrites the calls to the latter methods into calls dispatched to a copy of `this` (e.g., see the `_this` receiver in Fig. 1, lines 28–29). This enables IMMUTATOR to correctly refactor code that invokes library methods.

Escape analysis [7, 29] determines if an object escapes the current context. So far, the primary applications of this analysis were to determine whether (i) an object allocated inside a function does not escape and thus can be allocated on the stack, and (ii) whether an object is only accessed by one single thread, thus any synchronizations on that object can be removed. There are two major differences between these algorithms and IMMUTATOR's entering and escaping analysis. First, the search scope is different. We are only interested in detecting escaped objects that belong to the transitive state of the target class, thus needing to be cloned. Second, we are only interested on a demand-driven escape analysis, i.e., we do not want to perform an expensive whole program analysis, but only an analysis on the boundary methods of the target class.

Refactoring The related work in the area of automated refactoring is too long to cite it all. Although the traditional usages of refactorings are in the area of improving the code design, the more recent work has expanded the area with new usages. Tansey and Tilevich [26] present a refactoring for upgrading older framework-

based applications to use new framework versions based on annotations (for example from JUnit's 3.0 code conventions to JUnit's 4 annotations). More recently, we have used refactoring [9, 10] to retrofit parallelism into sequential applications via concurrent libraries. In the same spirit, Wloka et al. [30] present a refactoring for replacing global state with thread local state, which is an enabling refactoring for parallelism. Our current refactoring for immutability fits in the same category of enabling parallelism.

9. Conclusions

Programmers use immutability for both sequential programming and parallel programming. Although some classes are designed from the beginning to be immutable, other classes are retrofitted with immutability. Manually refactoring for immutability is tedious and error-prone.

Our refactoring tool, IMMUTATOR, automates the analysis and transformations required to make a class immutable. Controlled experiments and case studies show the IMMUTATOR is faster and safer than programmers.

Acknowledgments

This work was funded by the Universal Parallel Computing Research Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by Intel Corporation and Microsoft Corporation.

References

[1] T.j. watson libraries for analysis (wala). URL `http://wala.sourceforge.net/wiki/index.php`.

[2] L. O. Andersen. *Program Analysis and Specialization of the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[3] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29–41, 1979.

[4] D. Bäumer, D. Riehle, W. Siberski, C. Lilienthal, D. Megert, K.-H. Sylla, and H. Züllighoven. Want value objects in java? *Ubilab Technical Report*, 1998.

[5] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[6] J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001.

[7] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, and S. Midkiff. Escape analysis for java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.

[8] D. Dig. *Automated Upgrading of Component-based Applications*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[9] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 397–407, 2009.

[10] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: refactoring for loop parallelism in java. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 793–794, 2009.

[11] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Trans. Softw. Eng.*, 29(7):665–670, 2003.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[13] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[14] Immutator. Homepage. `http://refactoring.info/tools/Immutator`.

[15] Java Map. API Specification. `http://java.sun.com/javase/6/docs/api/java/util/Map.html`.

[16] Java String Interning. API Specification. `http://java.sun.com/javase/6/docs/api/java/lang/String.html#intern()`.

[17] JHotDraw framework. Homepage. `http://www.jhotdraw.org/`.

[18] D. Lea. *Concurrent Programming In Java*. Addison-Wesley, second edition, 2000.

[19] D. Marinov and R. O'Callahan. Object equality profiling. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, (OOPSLA)*, pages 313–325, 2003.

[20] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, 2002.

[21] D. Riehle. Value object. *Proceedings of the 2006 Conference on Pattern Languages of Programming*, 2006.

[22] A. Rountev. Precise identification of side-effect-free methods in java. *Proceedings of the 20th IEEE International Conference on Software Mainenance*, 2004.

[23] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[24] A. Salcianu and M. C. Rinard. Purity and side effect analysis for java programs. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference (VMCAI)*, pages 199–215, 2005.

[25] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for java. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, 2006.

[26] W. Tansey and E. Tilevich. Annotation refactoring: inferring upgrade transformations for legacy applications. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 295–312, 2008.

[27] D. Thomas. Functional programming – crossing the chasm? *Journal of object technology*, 2009.

[28] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2005.

[29] J. Whaley and M. C. Rinard. Compositional pointer and escape analysis for java programs. In *OOPSLA*, pages 187–206, 1999.

[30] J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *ESEC/FSE '09: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 173–182, 2009.

[31] X. Zheng and R. Rugina. Demand-driven alias analysis for c. *SIGPLAN Not.*, 43(1):197–208, 2008.

[32] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and reference immutability using java generics. *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.